IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re application of:	John M. Adolphson, et al.	:	Date: March 30, 2007
Group Art Unit:	2192	:	IBM Corporation
Examiner:	C. Kendall	:	Intellectual Property Law
Serial No.:	10/616,547	:	Dept. 917, Bldg. 006-1
Filed:	July 10, 2003	:	3605 Highway 52 North
Title:	METHOD AND APPARATUS FOR GENERATING COMPUTER PROGRAMMING CODE SELECTIVELY OPTIMIZED FOR EXECUTION PERFORMANCE AND NOT OPTIMIZED FOR SERVICEABILITY	:	Rochester, MN 55901

Commissioner for Patents
P.O. Box 1450
Alexandria, VA 223313-1450

APPEAL BRIEF IN SUPPORT OF APPEAL
FROM THE PRIMARY EXAMINER TO THE BOARD OF APPEALS

Sir:

This is an appeal of a Final Rejection under 35 U.S.C. §103(a) of claims 1-19 of Application Serial No. 10/616,547, filed December July 10, 2003. This brief is submitted pursuant to a Notice of Appeal filed January 31, 2007, as required by 37 C.F.R. §1.192.

1. Real Party in Interest

International Business Machines Corporation of Armonk, NY, is the real party in interest. The inventors assigned their interest as recorded on July 10, 2003, on Reel 014273, Frame 0962.

2. Related Appeals and Interferences

There are no related appeals nor interferences pending with this application.

3. Status of Claims

Claims 1-19 are pending and stand finally rejected. The claims on appeal are set forth in the Appendix of Claims

4. Status of Amendments

No amendments were submitted following Final Rejection.

5. Summary of Claimed Subject Matter

The invention herein relates to the use of optimizing compilers for producing executable computer programming code. Independent claims 1 and 13 recite respectively a method and a computer program product for compiling code, in which "selective optimization data" with respect to each of a plurality of discrete component portions of the source code is used to determine whether to perform optimizations during compilation. Independent claim 8 recites a method for compiling code, in which "debug activity data" is used to make selective optimization determinations during compilation, but does not recite that the debug activity data is with respect to discrete component portions.

An optimizing compiler has the capability to selectively optimize (i.e., to automatically turn optimization on or off, and to do so for selective portions of a code module being compiled) (Spec. p. 5, lines 13-18; p. 22, lines 17-19). Preferably, the compiler can selectively optimize at the granularity of a functional block, such as a procedure, function or method, although a different granularity could be used (Spec. p. 5, lines 19-23; p. 14, lines 22-25). Selective optimization data is generated with respect to the different portions of a compilable source module (Spec. p. 6, lines 17-24; p. 14, line 19 - p. 15, line 6; Fig. 3). Preferably, the selective optimization data includes both usage profile data (indicating frequency of use of each of multiple portions of the code module being compiled), and debug history data (indicating level of debug activity associated with different code portions) (Spec. p. 6, lines 17-28; p. 12, lines 2-19; p. 14, line 19 - p. 16, line 21; Fig. 3). Per claims 1 and 13, with respect to each portion, the compiler automatically makes a determination whether to optimize the respective portion based on the selective optimization data (Spec. p. 6, lines 1-10; p. 24, line 2 - p. 25, line 7; Figs 5A & 5B). Per claim 8, the compiler automatically makes multiple selective optimization determinations based on the debug data. (Spec. p. line 17-28; p. 24, line 2 - p. 25, line 7; Figs. 5A & 5B)

6. Grounds of Rejection To Be Reviewed on Appeal

Claims 1-5, 7, 8. 13-17 and 19 are finally rejected under 35 U.S.C. §103(a) as unpatentable over Smith et al. (US 6,311,324) in view of Chambers et al.(US 6,427,234). Claims 6 and 18 are rejected under 35 U.S.C. §103(a) as unpatentable over *Smith* and *Chambers*, further in view of Blume (US 6,223,337). Claims 9 and 11 are rejected under 35 U.S.C. §103(a) as unpatentable over *Smith* and *Chambers*, further in view of Hunt (US 6,499,137). Claims 10 and 12 are rejected under 35 U.S.C. §103(a) as unpatentable over

Smith, *Chambers*, and *Hunt*, further in view of Bates et al. (US 6,311,324). The only issues in this appeal are whether the claims are prima facie obvious over the cited references.

7. Argument

Appellants contend that the Examiner failed to establish adequate grounds of rejection for the following reasons:

I. The Examiner improperly rejected independent claims 1 and 13 under 35 U.S.C. §103(a) because neither *Smith* nor *Chambers* (nor the tertiary references), considered alone or in combination, discloses the key features of appellant's independent claims 1 and 13, i.e., generating separate selective optimization data for discrete code portions, and selectively determining whether to optimize code portions based on the selective optimization data. [page 7 below]

II. The Examiner improperly rejected independent claim 8 under 35 U.S.C. §103(a) because neither *Smith* nor *Chambers* (nor the tertiary references), considered alone or in combination, discloses the key features of independent claim 8, i.e., the use of debug history data to make selective optimization determinations in a compilation process. [page 13 below]

Overview of Invention

A brief overview of appellants' invention in light of existing art will be helpful in appreciating the issues herein. Optimizing compilers are well known in the art, and optimization of a compiled module is often a user-specified option in a compiler. Conventionally, a non-optimizing compiler (or a compiler in which optimization has been turned off by the user) sequentially converts each source code statement to a respective set of one or more executable instructions, so that there is a one-to-one correspondence between source code statements and sets of executable code instructions.

An optimizing compiler, on the other hand, has the intelligence to perform certain optimizations of the source code stream. For example, partial arithmetic products may be stored and re-used in later statements, data might be held in registers without storing and re-loading, some operations might re-ordered for greater efficiency, etc. Some of these optimizations are relatively simple, while others can be very complex.

Optimization produces code which is more performance efficient. However, there is a drawback to optimization. As a result of the various modifications made by the optimizing compiler, there is no one-to-one correspondence between source code statements and sets of executable code instructions produced by the optimizing process. This makes it significantly more difficult to debug optimized code, and it is said that ***optimized code is less serviceable***. For example, standard debug tools can trace events occurring during execution, but when there is little correspondence between the executable instructions and the source code, a trace of execution events is often meaningless information. As a result, it is common practice for code developers to generate non-optimized code during the development process, when there is a significant need to debug code, and to generate optimized code for release to customers only after the development process is deemed sufficiently complete.

But despite the best efforts of programmers, bugs will inevitably creep into the code that gets shipped to customers, i.e. the optimized code. Generally, it is these bugs that are the most difficult to detect and diagnose, for the simple ones are corrected during the development process. Often, such bugs only manifest themselves intermittently, and are difficult to reproduce. Since the customer has optimized code, it is generally difficult or impossible to diagnose the problem from customer data in the field. It is often necessary for developers to reproduce the bug in the laboratory using non-optimized code,

or failing that, to load non-optimized code on the customer's system in the field and wait for the bug to re-appear.

Appellants made this observation. Generally, a relatively small portion of code accounts for the bulk of execution time. If this small portion is optimized for performance, and the remaining code is non-optimized (and therefore readily serviceable using conventional tools), ***the resulting compiled program module will be almost as fast as a fully optimized program module, and almost as serviceable as one in which no optimizations are performed***. Appellants therefore propose to identify portions of the code for selective optimization using selective optimization data, and optimize only those portions.

Therefore a significant feature of appellants' invention is that optimization is automatically and selectively turned on and off based on selective optimization data, and specifically for different code portions. Selective optimization data can include various things, but preferably includes debug history data, i.e. data indicating the level of debug activity for different code portions, from which the probability of future debug activity may be inferred. Selective optimization data may further include frequency of use data.

It is well known in the art to use empirical profile data in the compilation process, specifically to optimize code. But this data is used to determine ***how to optimize code.*** It is assumed that the entire code module will be optimized, but certain decisions are made by the compiler based on profile data. E.g., a limited number of high-speed registers is assigned to most frequently used variables as determined by profiling data. Appellants propose to use empirical data to determine ***whether to optimize selective code portions***. There is a subtle but crucial difference.

I. The Examiner improperly rejected independent claims 1 and 13 under 35 U.S.C. §103(a) because neither *Smith* nor *Chambers* (nor the secondary references), considered alone or in combination, discloses the key features of appellant's independent claims 1 and 13, i.e., generating separate selective optimization data for discrete code portions, and selectively determining whether to optimize code portions based on the selective optimization data.

In order to support a rejection for obviousness, there must be some suggestion in the art to combine the references in such a manner as to form each and every element of appellants' claimed invention. It is not sufficient that a suggestion may exist to combine the references, if such a combination does not meet the limitations of appellants' claims without some further non-obvious modification. Assuming arguendo that a suitable suggestion exists in the art to combine the references, the hypothetical combination fails to teach or suggest the significant claimed features of appellants' invention, and therefore the rejection was improper.

Smith, the primary reference cited herein, discloses a tuning advisor for recommending that the programmer consider performing certain improvements to the source code. *Smith* sometimes refers to these source code improvements as "optimizations", but is should be clear that these are not optimizations performed by an automated compiler. They are source code improvements performed by the programmer. As disclosed in *Smith*, an automated compiler must make certain conservative assumptions about the behavior of the program in order to ensure that any executable code generated by the compiler will produce a logically correct result. Sometimes, an automated compiler can identify potential optimizations, but because it makes these conservative assumptions, it can not verify that the optimization will produce a logically correct result. By alerting the programmer to such potential optimizations, the programmer is enabled to make them him/herself, i.e., by appropriate manual modification of the source code. *Smith* is only a tool for aiding the programmer in

writing more efficient code. It does not disclose any change to the automated compilation process itself.

Chambers discloses an automated run-time compiler which has the capability to compile selective portions of a computer program at run time for better performance. As disclosed in *Chambers*, there are certain code functions or segments which perform more efficiently if compiled at run-time. Generally, these are value-specific portions of a program, in which run-time computed values of certain variables and data structures which do not change during execution are be used by the compiler for greater efficiency of execution. In accordance with *Chambers*, a programmer inserts certain compiler directives in the source code to selectively compile some portions of the code statically, and other portions at run-time (dynamically). The compiler directives may be conditioned on the values of run-time conditions, the conditional statements being evaluated at run-time to determine whether to re-compile the source with run-time values.

The Examiner apparently reasons that *Smith* discloses use of "selective optimization data" for making optimization determinations, and that *Chambers* discloses selective run-time compilation (which the Examiner equates to "optimization") of discrete code portions, and hence the combination of the two references suggests appellants' claimed invention.

The problem with this line reasoning is that the Examiner is identifying abstract properties of the techniques disclosed in the two references, and combining these abstract properties to find the same abstract properties as appellants' invention. Appellants' invention is not a collection of abstract properties without any interplay among them, but

a specific set of method steps (or steps performed by a computer program product) to produce a result.

Appellants' representative claim 1 recites:

1. A method for compiling computer programming code, comprising the steps of:

generating a compilable source module, said source module containing a plurality of discrete component portions;

generating selective optimization data, *said selective optimization data including a plurality of selective optimization data portions, each of said plurality of selective optimization data portions corresponding to a respective component portion of said plurality of discrete component portions*; and

compiling said compilable source module with an automated compiler, wherein said compiling step comprises the following *steps performed by said automated compiler*:

(a) with respect to each of said plurality of discrete component portions, *selectively determining whether to optimize the respective discrete component portion using said selective optimization data portion* corresponding to the respective discrete component portion;

(b) with respect to at least one discrete component portion of a first subset of said plurality of discrete component portions, said first subset containing the discrete component portions for which said selectively determining step determined to optimize the respective discrete component portion, *performing at least one optimization upon the respective discrete component portion responsive to said selectively determining step*; and

(c) with respect to at least one discrete component portion of a second subset of said plurality of discrete component portions, said second subset containing the discrete component portions for which said selectively determining step determined not to optimize the respective discrete component portion, *compiling the respective discrete component portion without performing at least one optimization which said automated compiler has the capability to automatically perform* on the respective discrete component portion.

[emphasis added]

Independent claim 13 is a program product claim which contains analogous limitations to the italicized language above.

First and foremost, appellants' claims 1 and 13 recite something ***performed by an automated compiler***. *Smith* discloses a technique for generating advice to a programmer, but ***the programmer manually alters the source code*** in response to that advice. These are referred to in *Smith* as "optimizations", and in a general sense they are, but they are not "optimizations" as used in appellants' claims, i.e., optimizations performed by an automated compiler as part of the compilation process.

If the hypothetical combination of *Smith* and *Chambers* is assumed, it is useful to ask, just what is it that is suggested by such a combination? *Smith* discloses a system for giving programming advice to the programmer, who then manually alters source code to make it more efficient. *Chambers* discloses a technique for selectively compiling certain code portions at run-time. Manifestly, the combination of the two references amounts to a technique for advising the programmer to manually alter source code, manually making the alterations (as in *Smith*), and subsequently compiling selective portions at run-time (as in *Chambers*). ***But such a hypothetical combination does not meet the limitations of appellants' claims.***

In citing *Chambers*, the Examiner is equating compilation with optimization. In a general sense, a compilation is an optimization, for it does something to further a process for which the code was written. But "optimization" as used in the context of a compilation process means something quite different. It is a well established usage in the art of compilers that "optimization" means doing something beyond ordinary compilation, in which a certain level of intelligence in the compiler is able to change the order of operations, delete some operations, alter others, from what is specified in the source code, so that the result is something other than a fairly direct translation of source code.

Appellants are not asking that limitations from the specification be imported into the claims, but only that the claim as a whole be given its logical and necessary meaning. It is true that appellants' claims do not recite any particular optimization (various optimizations are known in the compiler art). But the claims do recite optimization ***in the context of a compilation process***, i.e.:

> compiling said compilable source module with an automated compiler, wherein said compiling step comprises the following steps . . .
> (b) ... performing at least one optimization upon the respective discrete component portion responsive to said selectively determining step; and
> (c) ... compiling the respective discrete component portion without performing at least one optimization which said automated compiler has the capability to automatically perform on the respective discrete component portion. [claim 1].

These steps recite selectively optimizing something ***within the compilation***. Step (c) explicitly recites a step of compiling without performing the optimization. If, as the Examiner seems to suggest, an "optimization" reads upon a compilation itself, then the above steps make no sense. The only way in which the recited steps can make any sense is if "optimization" is understood to mean any of various so-called optimizing techniques as are known in the art of automated compilers, and which are performed within the compilation.

Neither *Smith* nor *Chambers* discloses nor suggests selectively applying such known optimization techniques to different portions of the code during an automated compilation.

It is instructive to reflect a moment just what it is appellants' invention is meant to accomplish. Appellants' invention is intended to produce compiled executable code which is easier to maintain in the field. It is expected that most of the code, and in

particular the more recent modifications to it, will be non-optimized when compiled with appellant's technique. Thus, ***the product of appellants' compilation process is inherently less efficient*** than conventional optimized code. This is a deliberate decision made by appellants, to improve the serviceability of code in the field. If appropriate selective criteria are used for determining which portions of code to optimize and which not to optimize, as taught by appellants' disclosure, it should be possible to produce compiled code which is still reasonably efficient, yet far more serviceable than fully optimized code.

The primary references cited herein are aimed at doing one thing and one thing only: maximizing performance efficiency. A system such as appellants', which deliberately sacrifices performance efficiency to achieve some other goal, is therefore at odds with the teachings of the references, and these references teach away from appellants' invention.

There is nothing whatsoever in either *Smith* or *Chambers* to teach or suggest the basic problem of serviceability of fully optimized code, or a solution involving optimizing of selective portions. These references aim only to increase performance efficiency, without regard to serviceability. Essentially, the Examiner has arbitrarily combined high-level attributes of the references to build a case of obviousness. The Examiner does not even combine elements or steps, because the critical elements don't exist in either reference, as explained above. But even ignoring this discrepancy, one must ultimately ask this question: ***Where is the motivation shown in the art*** to do what appellants claim, i.e., to optimize only selective portions of code? This motivation can not be found in references which teach only how to improve performance efficiency. It is found in only one place: appellants' disclosure.

The various tertiary references similarly fail to teach or suggest the key limitations of appellants' claims as discussed above. *Blume* discloses a compiler in which optimization can be turned off for the entire code module, as is well known in the art, but does not teach or suggest turning compilation off with respect to selective code portions. *Hunt* discloses a method for dynamic linking, and is cited to show the use of counters to record debug events, but similarly does not teach or suggest appellants' essential features. *Bates* discloses the use of conditional breakpoints for debug, and is cited to show the use of variable visualization counters to record debug events. Like *Hunt*, it too does not teach or suggest the essential features of appellants' invention.

For all the reasons explained above, the combination of the cited references (even assuming it was proper to combine the references) fails to teach or suggest the critical limitations of appellants' claims, and the rejections of claims 1 and 13 (and there dependent claims) were erroneous.

II. The Examiner improperly rejected independent claim 8 under 35 U.S.C. §103(a) because neither *Smith* nor *Chambers* (nor the tertiary references), considered alone or in combination, discloses the key features of independent claim 8, i.e., the use of debug history data to make selective optimization determinations in a compilation process.

Independent claim 8 is broader in certain respects than claims 1 and 13, and narrower in others.[1] In particular, independent claim 8 recites that a particular form of data is used for making selective optimization determinations by the compiler, i.e.,

[1] For example, independent claim 8 is less explicit with respect to the correspondence between data used for making selective determinations and portions of the code. Claims 1 and 13 recite that "selective optimization data" includes multiple portions corresponding to respective portions of the code. Claim 8 recites only that the debug history data is with respect the compilable source module. There are additional differences.

"debug activity data". The purpose of using "debug activity data" is that code portions which have been recently changed or which have a history of frequent debug activity are more likely to require service in the future, and hence should probably not be optimized. Claim 8 is patentable over the cited art because none of the references even remotely shows the use of debug activity data in the compilation process.

Appellants' independent claim 8 recites:

> 8. A method for compiling computer programming code, comprising the steps of:
>
> generating a compilable source module;
>
> *generating debug activity data with respect to said compilable source module*; and
>
> compiling said compilable source module with an automated compiler, wherein said compiling step comprises the following steps performed by said automated compiler:
>
> (a) *making a plurality of selective optimization determinations with respect to said compilable source module using said debug activity data*; and
>
> (b) performing at least one respective optimization step responsive to each said selective optimization determination. [emphasis added]

In his rejection, the Examiner cites the following passage from *Smith* as a teaching of using debug activity data:

> CTA [C tuning advisor] can detect a range of line numbers on which to give advice, even when a user double clicks on a single line of code, by examining the control flow structure of the program. So CTA can automatically detect the context on which to give advice when the user just double clicks on a single source code line containing the hotspot. [col 4, lines 21-26]

The quoted passage merely says that the user can identify a source code statement, and that responsive thereto the CTA determines the range of statements (such as a basic block) which includes that statement. This passage does not by any stretch of the imagination disclose the use of "debug activity data". It does not record debug activity, it

does not generate anything as a result of debug activity, and it does not use any data for debug activity. It is unclear what the Examiner considers the "data". But even if one were to find "debug activity data" somewhere in this passage, there is a more fundamental problem. There is nothing which discloses or suggests using this data in an automated compiler to make selective optimization determinations.

As for the proposed combination of *Smith* and *Chambers*, appellant makes the same observations made previously with respect to claims 1 and 13. *Chambers* does not disclose a system which "optimizes" selective code segments; it discloses a system which compiles selective code segments at run-time. As explained above, appellants use the term "optimization" within the context of a compilation process. It is necessarily something other than compilation itself, or the limitations of the claims have no meaning. Appellants' claim 8 recites that ***compilation*** of the source code module ***comprises*** making a plurality of selective optimization determinations. If compilation is equated with optimization, then this limitation makes no sense, and is essentially read out of the claims. Appellant submits that such a reading of the word "optimization", in the context of claim 8, is unreasonable.

The tertiary references do indeed disclose the existence of debug activity data, but there is no disclosure or suggestion that such data be used for making selective optimization determinations in a compilation process.

For all of these reasons, the cited references do not teach or suggest the use of "debug activity data" to make "selective optimization determinations" in an automated compiler, and the Examiner's rejections of claim 8 (and its dependent claims) were improper.

8. Summary

Appellants disclose and claim a novel and unobvious technique for compiling computer programming code, in which selective portions of code are deliberately not optimized in order to improve serviceability of the resultant compiled executable code. The resultant product is inherently less efficient that fully optimized code, but is a reasonable compromise between efficiency and serviceability. The primary references disclose techniques for improving the efficiency only of code, and this direction teaches away from a technique such as appellants'. There is no suggestion of the problem addressed by appellants, nor the solution which they propose to overcome it.

For all the reasons stated herein, the rejections for obviousness were improper, and appellants respectfully requests that the Examiner's rejections of the claims be reversed.

Date: March 30, 2007

Respectfully submitted,

JOHN M ADOLPHSON, et al.

By_____

Roy W. Truelson, Attorney
Registration No. 34,265
(507) 202-8725 (Cell) (507) 289-6256 (Office)

From: IBM Corporation
Intellectual Property Law
Dept. 917, Bldg. 006-1
3605 Highway 52 North
Rochester, MN 55901

APPENDIX OF CLAIMS

1. A method for compiling computer programming code, comprising the steps of:

generating a compilable source module, said source module containing a plurality of discrete component portions;

generating selective optimization data, said selective optimization data including a plurality of selective optimization data portions, each of said plurality of selective optimization data portions corresponding to a respective component portion of said plurality of discrete component portions; and

compiling said compilable source module with an automated compiler, wherein said compiling step comprises the following steps performed by said automated compiler:

(a) with respect to each of said plurality of discrete component portions, selectively determining whether to optimize the respective discrete component portion using said selective optimization data portion corresponding to the respective discrete component portion;

(b) with respect to at least one discrete component portion of a first subset of said plurality of discrete component portions, said first subset containing the discrete component portions for which said selectively determining step determined to optimize the respective discrete component portion, performing at least one optimization upon the respective discrete component portion responsive to said selectively determining step; and

(c) with respect to at least one discrete component portion of a second subset of said plurality of discrete component portions, said second subset containing the discrete component portions for which said selectively determining step determined not to optimize the respective discrete component portion, compiling the respective discrete component portion without performing at least one

optimization which said automated compiler has the capability to automatically perform on the respective discrete component portion.

2. The method for compiling computer programming code of claim 1, wherein said component portion is a procedure.

3. The method for compiling computer programming code of claim 1, wherein said selective optimization data comprises data concerning debug activity occurring with respect to each of said plurality of discrete component portions.

4. The method for compiling computer programming code of claim 1, wherein said selective optimization data comprises data concerning execution time with respect to each of said plurality of discrete component portions.

5. The method for compiling computer programming code of claim 1, wherein said selective optimization data comprises a plurality of optimization flags, each optimization flag corresponding to a respective component portion of said plurality of discrete component portions.

6. The method for compiling computer programming code of claim 1, wherein said compiling step comprises, with respect to a first discrete component portion, but not with respect to all said discrete component portions, generating alternative compiled versions of the first discrete component portion, wherein a first alternative version of said first discrete component portion is produced by performing a first optimization, and a second alternative version of said first discrete component portion is produced without performing said first optimization.

7. The method for compiling computer programming code of claim 1, wherein:

said step (a) comprises, with respect to each of said plurality of discrete component portions, determining a corresponding optimization level from among at least three distinct optimization levels, wherein the optimization performed at a first level are greater than the optimizations performed at a second level, and the optimizations performed at a second level are greater than the optimizations, if any, performed at a third level; and

said step (b) comprises performing optimization on each respective discrete component portion according to its corresponding optimization level.

8. A method for compiling computer programming code, comprising the steps of:

generating a compilable source module;

generating debug activity data with respect to said compilable source module; and

compiling said compilable source module with an automated compiler, wherein said compiling step comprises the following steps performed by said automated compiler:

(a) making a plurality of selective optimization determinations with respect to said compilable source module using said debug activity data; and

(b) performing at least one respective optimization step responsive to each said selective optimization determination.

9. The method for compiling computer programming code of claim 8, wherein said debug activity data comprises a plurality of counters, each counter being incremented upon the occurrence of a corresponding debug event.

10. The method for compiling computer programming code of claim 10, wherein each counter is incremented upon the occurrence of a corresponding debug event by an amount derived from a user weighting factor associated with a user on whose behalf the debug event occurs.

11. The method for compiling computer programming code of claim 10, wherein said debug activity data comprises a plurality of break-point counters, each break-point counter corresponding to a respective portion of said compilable source module, each break-point counter being incremented upon the occurrence of a break point triggered within the corresponding respective portion of said compilable source module.

12. The method for compiling computer programming code of claim 10, wherein said debug activity data comprises a plurality of variable visualization counters, each variable visualization counter corresponding to a respective variable used in said compilable source module, each variable visualization counter being incremented upon the occurrence of a user directed visualization of the corresponding variable during debug activity..

13. A computer program product for compiling computer programming code, comprising:

a plurality of executable instructions recorded on tangible signal-bearing media, wherein said instructions, when executed by at least one processor of a digital computing device, cause the device to perform the steps of:

receiving a compilable source module, said source module containing a plurality of discrete component portions;

receiving selective optimization data, said selective optimization data including a plurality of selective optimization data portions, each of said plurality of selective optimization data portions corresponding to a respective component portion of said plurality of discrete component portions; and

compiling said compilable source module, wherein said compiling step comprises:

(a) with respect to each of said plurality of discrete component portions, selectively determining whether to optimize the respective discrete component portion using said selective optimization data portion corresponding to the respective discrete component portion;

(b) with respect to at least one discrete component portion of a first subset of said plurality of discrete component portions, said first subset containing the discrete component portions for which said selectively determining step determined to optimize the respective discrete component portion, performing at least one optimization upon the respective discrete component portion responsive to said selectively determining step; and

(c) with respect to at least one discrete component portion of a second subset of said plurality of discrete component portions, said second subset containing the discrete component portions for which said selectively determining step determined not to optimize the respective discrete component portion, compiling the respective discrete component portion without performing at least one optimization which said computer program product has the capability to automatically perform on the respective discrete component portion.

14. The computer program product for compiling computer programming code of claim 13, wherein said component portion is a procedure.

15. The computer program product for compiling computer programming code of claim 13, wherein said selective optimization data comprises data concerning debug activity occurring with respect to each of said plurality of discrete component portions.

16. The computer program product for compiling computer programming code of claim 13, wherein said selective optimization data comprises data concerning execution time with respect to each of said plurality of discrete component portions.

17. The computer program product for compiling computer programming code of claim 13, wherein said selective optimization data comprises a plurality of optimization flags, each optimization flag corresponding to a respective component portion of said plurality of discrete component portions.

18. The computer program product for compiling computer programming code of claim 13, wherein said compiling step comprises, with respect to a first discrete component portion, but not with respect to all said discrete component portions, generating alternative compiled versions of the first discrete component portion, wherein a first alternative version of said first discrete component portion is produced by performing a first optimization, and a second alternative version of said first discrete component portion is produced without performing said first optimization.

19. The computer program product for compiling computer programming code of claim 13, wherein:

said step (a) comprises, with respect to each of said plurality of discrete component portions, determining a corresponding optimization level from among at least three distinct optimization levels, wherein the optimization performed at a first level are greater than the optimizations performed at a second level, and the optimizations performed at a second level are greater than the optimizations, if any, performed at a third level; and

said step (b) comprises performing optimization on each respective discrete component portion according to its corresponding optimization level.

APPENDIX OF EVIDENCE

No evidence is submitted.

APPENDIX OF RELATED PROCEEDINGS

There are no related proceedings.